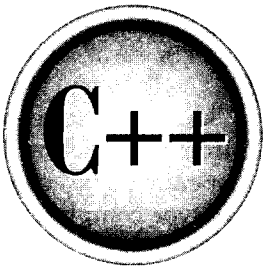
The Complete Reference



Chapter 37

The Numeric Classes

One of the features added during the standardization of C++ is the numeric class library. These classes aid in the development of numerical programs. Several of the member functions of these classes parallel the stand-alone functions inherited from the C library. The difference is that many of the numeric functions described here operate on objects of type **valarray**, which is essentially an array of values, or on objects of type **complex**, which represent complex numbers. By including the numeric classes, Standard C++ has expanded the scope of programming tasks to which it can be conveniently applied.

The complex Class

The header **<complex>** defines the **complex** class, which represents complex numbers. It also defines a series of functions and operators that operate on objects of type **complex**.

The template specification for **complex** is shown here:

```
template <class T> class complex
```

Here, **T** specifies the type used to store the components of a complex number. There are three predefined specializations of **complex**:

```
class complex<float>
class complex<double>
class complex<long double>
```

The **complex** class has the following constructors:

```
complex(const T &real = T( ), const T &imaginary = T( ));
complex(const complex &ob);
template <class T1> complex(const complex<T1> &ob);
```

The first constructs a **complex** object with a real component of *real* and an imaginary component of *imaginary*. These values default to zero if not specified. The second creates a copy of *ob*. The third creates a **complex** object from *ob*.

The following operations are defined for **complex** objects:

+	-	*	/
-=	+=	/=	*=
=	==	!=	

The nonassignment operators are overloaded three ways. Once for operations involving a **complex** object on the left and a scalar object on the right, again for operations involving a scalar on the left and a **complex** object on the right, and finally for operations involving two **complex** objects. For example, the following types of operations are allowed:

```
complex_ob + scalar
scalar + complex_ob
complex_ob + complex_ob
```

Operations involving scalar quantities affect only the real component.

Two member functions are defined for **complex**: **real( )** and **imag( )**. They are shown here:

```
T real( ) const;
T imag( ) const;
```

The **real( )** function returns the real component of the invoking object, and **imag( )** returns the imaginary component. The functions shown in Table 37-1 are also defined for **complex** objects.

Here is a sample program that demonstrates **complex**.

```
// Demonstrate complex.
#include <iostream>
#include <complex>
using namespace std;

int main()
{
  complex<double> cmpx1(1, 0);
  complex<double> cmpx2(1, 1);

  cout << cmpx1 << " " << cmpx2 << endl;

  complex<double> cmpx3 = cmpx1 + cmpx2;
  cout << cmpx3 << endl;

  cmpx3 += 10;
  cout << cmpx3 << endl;

  return 0;
}
```

Its output is shown here:

```
(1,0) (1,1)
(2,1)
(12,1)
```

Function	Description
template <class T> T abs(const complex<T> &*ob*);	Returns the absolute value of *ob*.
template <class T> T arg(const complex<T> &*ob*);	Returns the phase angle of *ob*.
template <class T> complex<T> conj(const complex<T> &*ob*);	Returns the conjugate of *ob*.
template <class T> complex<T> cos(const complex<T> &*ob*);	Returns the cosine of *ob*.
template <class T> complex<T> cosh(const complex<T> &*ob*);	Returns the hyperbolic cosine of *ob*.
template <class T> complex<T> exp(const complex<T> &*ob*);	Returns the e^{ob}.
template <class T> T imag(const complex<T> &*ob*);	Returns the imaginary component of *ob*.
template <class T> complex<T> log(const complex<T> &*ob*);	Returns the natural logarithm of *ob*.
template <class T> complex<T> log10(const complex<T> &*ob*);	Returns the base 10 logarithm of *ob*.
template <class T> T norm(const complex<T> &*ob*);	Returns the magnitude of *ob* squared.

Table 37-1. *Functions Defined for* ***complex***

Function	Description
template <class T> complex<T> polar(const T &*v*, const T &*theta*=0);	Returns a complex number that has the magnitude specified by *v* and a phase angle of *theta*.
template <class T> complex<T> pow(const complex<T> &*b*, int *e*);	Returns b^e.
template <class T> complex<T> pow(const complex<T> &*b*, const T &*e*);	Returns b^e.
template <class T> complex<T> pow(const complex<T> &*b*, const complex<T> &*e*);	Returns b^e.
template <class T> complex<T> pow(const T &*b*, const complex<T> &*e*);	Returns b^e.
template <class T> T real(const complex<T> &*ob*);	Returns the real component of *ob*.
template <class T> complex<T> sin(const complex<T> &*ob*);	Returns the sine of *ob*.
template <class T> complex<T> sinh(const complex<T> &*ob*);	Returns the hyperbolic sine of *ob*.
template <class T> complex<T> sqrt(const complex<T> &*ob*);	Returns the square root of *ob*.
template <class T> complex<T> tan(const complex<T> &*ob*);	Returns the tangent of *ob*.
template <class T> complex<T> tanh(const complex<T> &*ob*);	Returns the hyperbolic tangent of *ob*.

Table 37-1. *Functions Defined for* ***complex*** (continued)

The valarray Class

The header **<valarray>** defines a number of classes that support numeric arrays. The main class is **valarray**, and it defines a one-dimensional array of values. There are a wide variety of member operators and functions defined for it as well as a large number of nonmember functions. While the description of **valarray** that is given here will be sufficient for most programmers, those especially interested in numeric processing will want to study **valarray** in greater detail. One other point: Although **valarray** is very large, most of its operations are intuitive.

The **valarray** class has this template specification:

```
template <class T> class valarray
```

It defines the following constructors:

```
valarray( );
explicit valarray (size_t num);
valarray(const T &v, size_t num);
valarray(const T *ptr, size_t num);
valarray(const valarray<T> &ob);
valarray(const slice_array<T> &ob);
valarray(const gslice_array<T> &ob);
valarray(const mask_array<T> &ob);
valarray(const indirect_array<T> &ob);
```

Here, the first constructor creates an empty object. The second creates a **valarray** of length *num*. The third creates a **valarray** of length *num* initialized to *v*. The fourth creates a **valarray** of length *num* and initializes it with the elements pointed to by *ptr*. The fifth form creates a copy of *ob*. The next four constructors create a **valarray** from one of **valarray**'s helper classes.

The following operators are defined for **valarray**:

+	-	*	/
-=	+=	/=	*=
=	==	!=	<<
>>	<<=	>>=	^
^=	%	%=	~
!	|	|=	&
&=	[ ]		

These operators have several overloaded forms that are described in the accompanying tables.

The member functions and operators defined by **valarray** are shown in Table 37-2. The nonmember operator functions defined for **valarray** are shown in Table 37-3. The transcendental functions defined for **valarray** are shown in Table 37-4.

Function	Description
valarray<T> apply(T *func*(T)) const; valarray<T> apply(T *func*(const T &*ob*)) const;	Applies *func( )* to the invoking array and returns an array containing the result.
valarray<T> cshift(int *num*) const;	Left-rotates the invoking array *num* places. (That is, it performs a circular shift left.) Returns an array containing the result.
T max( ) const;	Returns the maximum value in the invoking array.
T min( ) const	Returns the minimum value in the invoking array.
valarray<T> &operator=(const valarray<T> &*ob*);	Assigns the elements in *ob* to the corresponding elements in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator=(const T &*v*);	Assigns each element in the invoking array the value *v*. Returns a reference to the invoking array.
valarray<T> &operator=(const slice_array<T> &*ob*);	Assigns a subset. Returns a reference to the invoking array.
valarray<T> &operator=(const gslice_array<T> &*ob*);	Assigns a subset. Returns a reference to the invoking array.
valarray<T> &operator=(const mask_array<T> &*ob*);	Assigns a subset. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray***

Function	Description
valarray<T> &operator=(const indirect_array<T> &*ob*);	Assigns a subset. Returns a reference to the invoking array.
valarray<T> operator+( ) const;	Unary plus applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator–( ) const;	Unary minus applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator~( ) const;	Unary bitwise NOT applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator!( ) const;	Unary logical NOT applied to each element in the invoking array. Returns the resulting array.
valarray<T> &operator+=(const T &*v*) const;	Adds *v* to each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator–=(const T &*v*) const;	Subtracts *v* from each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator/=(const T &*v*) const;	Divides each element in the invoking array by *v*. Returns a reference to the invoking array.
valarray<T> &operator*=(const T &*v*) const;	Multiplies each element in the invoking array by *v*. Returns a reference to the invoking array.
valarray<T> &operator%=(const T &*v*) const;	Assigns each element in the invoking array the remainder of a division by *v*. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** (continued)

Function	Description
valarray<T> &operator^=(const T &*v*) const;	XORs *v* with each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator&=(const T &*v*) const;	ANDs *v* with each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator|=(const T &*v*) const;	ORs *v* to each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator<<=(const T &*v*) const;	Left-shifts each element in the invoking array *v* places. Returns a reference to the invoking array.
valarray<T> &operator>>=(const T &*v*) const;	Right-shifts each element in the invoking array *v* places. Returns a reference to the invoking array.
valarray<T> &operator+=(const valarray<T> &*ob*) const;	Corresponding elements of the invoking array and *ob* are added together. Returns a reference to the invoking array.
valarray<T> &operator-=(const valarray<T> &*ob*) const;	The elements in *ob* are subtracted from their corresponding elements in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator/=(const valarray<T> &*ob*) const;	The elements in the invoking array are divided by their corresponding elements in *ob*. Returns a reference to the invoking array.
valarray<T> &operator*=(const valarray<T> &*ob*) const;	Corresponding elements of the invoking array and *ob* are multiplied together. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** (continued)

Function	Description
valarray<T> &operator%=(const valarray<T> &*ob*) const;	The elements in the invoking array are divided by their corresponding elements in *ob* and the remainder is stored. Returns a reference to the invoking array.
valarray<T> &operator^=(const valarray<T> &*ob*) const;	The XOR operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator&=(const valarray<T> &*ob*) const;	The AND operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator|=(const valarray<T> &*ob*) const;	The OR operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator<<=(const valarray<T> &*ob*) const;	Elements in the invoking array are left-shifted by the number of places specified in the corresponding elements in *ob*. Returns a reference to the invoking array.
valarray<T> &operator>>=(const valarray<T> &*ob*) const;	Elements in invoking array are right-shifted by the number of places specified in the corresponding elements in *ob*. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** (continued)

Function	Description
T &operator[ ] (size_t *indx*) ;	Returns a reference to the element at the specified index.
T operator[ ] (size_t *indx*) const;	Returns the value at the specified index.
slice_array<T> operator[ ](slice *ob*);	Returns the specified subset.
valarray<T> operator[ ](slice *ob*) const;	Returns the specified subset.
gslice_array<T> operator[ ](const gslice &*ob*);	Returns the specified subset.
valarray<T> operator[ ](const gslice &*ob*) const;	Returns the specified subset.
mask_array<T> operator[ ](valarray<bool> &*ob*);	Returns the specified subset.
valarray<T> operator[ ](valarray<bool> &*ob*) const;	Returns the specified subset.
indirect_array<T> operator[ ](const valarray<size_t> &*ob*);	Returns the specified subset.
valarray<T> operator[ ](const valarray<size_t> &*ob*) const;	Returns the specified subset.
void resize(size_t *num*, T *v* = T( ));	Resizes the invoking array. If elements must be added, they are assigned the value of *v*.
size_t size( ) const;	Returns the size (i.e., the number of elements) of the invoking array.
valarray<T> shift(int *num*) const;	Shifts the invoking array left *num* places. Returns an array containing the result.
T sum( ) const;	Returns the sum of the values stored in the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** (continued)

Function	Description
template <class T> valarray<T> operator+(const valarray<T> *ob*, const T &*v*);	Adds *v* to each element of *ob*. Returns an array containing the result.
template <class T> valarray<T> operator+(const T &*v*, const valarray<T> *ob*);	Adds *v* to each element of *ob*. Returns an array containing the result.
template <class T> valarray<T> operator+(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Adds each element in *ob1* to its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator–(const valarray<T> *ob*, const T &*v*);	Subtracts *v* from each element of *ob*. Returns an array containing the result.
template <class T> valarray<T> operator–(const T &*v*, const valarray<T> *ob*);	Subtracts each element of *ob* from *v*. Returns an array containing the result.
template <class T> valarray<T> operator–(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Subtracts each element in *ob2* from its corresponding element in *ob1*. Returns an array containing the result.
template <class T> valarray<T> operator*(const valarray<T> *ob*, const T &*v*);	Multiplies each element in *ob* by *v*. Returns an array containing the result.
template <class T> valarray<T> operator*(const T &*v*, const valarray<T> *ob*);	Multiplies each element in *ob* by *v*. Returns an array containing the result.
template <class T> valarray<T> operator*(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Multiplies corresponding elements in *ob1* by those in *ob2*. Returns an array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray***

Function	Description
template <class T> valarray<T> operator/(const valarray<T> *ob*, const T &*v*);	Divides each element in *ob* by *v*. Returns an array containing the result.
template <class T> valarray<T> operator/(const T &*v*, const valarray<T> *ob*);	Divides *v* by each element in *ob*. Returns an array containing the result.
template <class T> valarray<T> operator/(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Divides each element in *ob1* by its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator%(const valarray<T> *ob*, const T &*v*);	Obtains the remainder that results from dividing each element in *ob* by *v*. Returns an array containing the result.
template <class T> valarray<T> operator%(const T &*v*, const valarray<T> *ob*);	Obtains the remainder that results from dividing *v* by each element in *ob*. Returns an array containing the result.
template <class T> valarray<T> operator%(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Obtains the remainder that results from dividing each element in *ob1* by its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator^(const valarray<T> *ob*, const T &*v*);	XORs each element in *ob* with *v*. Returns an array containing the result.
template <class T> valarray<T> operator^(const T &*v*, const valarray<T> *ob*);	XORs each element in *ob* with *v*. Returns an array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray*** (continued)

Function	Description
template <class T> valarray<T> operator^(const valarray<T> *ob1*, const valarray<T> &*ob2*);	XORs each element in *ob1* with its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator&(const valarray<T> *ob*, const T &*v*);	ANDs each element in *ob* with *v*. Returns an array containing the result.
template <class T> valarray<T> operator&(const T &*v*, const valarray<T> *ob*);	ANDs each element in *ob* with *v*. Returns an array containing the result.
template <class T> valarray<T> operator&(const valarray<T> *ob1*, const valarray<T> &*ob2*);	ANDs each element in *ob1* with its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator|(const valarray<T> *ob*, const T &*v*);	ORs each element in *ob* with *v*. Returns an array containing the result.
template <class T> valarray<T> operator|(const T &*v*, const valarray<T> *ob*);	ORs each element in *ob* with *v*. Returns an array containing the result.
template <class T> valarray<T> operator|(const valarray<T> *ob1*, const valarray<T> &*ob2*);	ORs each element in *ob1* with its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator<<(const valarray<T> *ob*, const T &*v*);	Left-shifts each element in *ob* by the number of places specified by *v*. Returns an array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray*** (continued)

Function	Description
template <class T> valarray<T> operator<<(const T &*v*, const valarray<T> *ob*);	Left-shifts *v* the number of places specified by the elements in *ob*. Returns an array containing the result.
template <class T> valarray<T> operator<<(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Left-shifts each element in *ob1* the number of places specified by its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<T> operator>>(const valarray<T> *ob*, const T &*v*);	Right-shifts each element in *ob* the number of places specified by *v*. Returns an array containing the result.
template <class T> valarray<T> operator>>(const T &*v*, const valarray<T> *ob*);	Right-shifts *v* the number of places specified by the elements in *ob*. Returns an array containing the result.
template <class T> valarray<T> operator>>(const valarray<T> *ob1*, const valarray<T> &*ob2*);	Right-shifts each element in *ob1* the number of places specified by its corresponding element in *ob2*. Returns an array containing the result.
template <class T> valarray<bool> operator==(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] == *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator==(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* == *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator==(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] == *ob2*[i]. Returns a Boolean array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray*** (continued)

Function	Description
template <class T> valarray<bool> operator!=(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] != *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator!=(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* != *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator!=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] != *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] < *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* < *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] < *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<=(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] <= *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<=(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* <= *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator<=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] <= *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator>(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] > *v*. Returns a Boolean array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray*** (continued)

Function	Description
template <class T> valarray<bool> operator>(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* > *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator>(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] > *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator>=(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] >= *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator>=(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* >= *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator>=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] >= *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator&&(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] && *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator&&(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* && *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator&&(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] && *ob2*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator||(const valarray<T> *ob*, const T &*v*);	For every i, performs *ob*[i] || *v*. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator||(const T &*v*, const valarray<T> *ob*);	For every i, performs *v* || *ob*[i]. Returns a Boolean array containing the result.
template <class T> valarray<bool> operator||(const valarray<T> *ob1*, const valarray<T> &*ob2*);	For every i, performs *ob1*[i] || *ob2*[i]. Returns a Boolean array containing the result.

Table 37-3. *The Nonmember Operator Functions Defined for* ***valarray*** (continued)

Function	Description
template<class T> valarray<T> abs(const valarray<T> &*ob*);	Obtains the absolute value of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> acos(const valarray<T> &*ob*);	Obtains the arc cosine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> asin(const valarray<T> &*ob*);	Obtains the arc sine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> atan(const valarray<T> &*ob*);	Obtains the arc tangent of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> atan2(const valarray<T> &*ob1*, const valarray<T> &*ob2*);	For all i, obtains the arc tangent of *ob1*[i] / *ob2*[i] and returns an array containing the result.
template<class T> valarray<T> atan2(const T &*v*, const valarray<T> &*ob*);	For all i, obtains the arc tangent of *v* / *ob1*[i] and returns an array containing the result.
template<class T> valarray<T> atan2(const valarray<T> &*ob*, const T &*v*);	For all i, obtains the arc tangent of *ob1*[i] / *v* and returns an array containing the result.
template<class T> valarray<T> cos(const valarray<T> &*ob*);	Obtains the cosine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> cosh(const valarray<T> &*ob*);	Obtains the hyperbolic cosine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> exp(const valarray<T> &*ob*);	Computes exponential function for each element in *ob* and returns an array containing the result.

Table 37-4. *Transcendental Functions Defined for* ***valarray***

Function	Description
template<class T> valarray<T> log(const valarray<T> &*ob*);	Obtains the natural logarithm of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> log10(const valarray<T> &*ob*);	Obtains the common logarithm of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> pow(const valarray<T> &*ob1*, const valarray<T> &*ob2*);	For all i, computes $ob1[i]^{ob2[i]}$ and returns an array containing the result.
template<class T> valarray<T> pow(const T &*v*, const valarray<T> &*ob*);	For all i, computes $v^{ob[i]}$ and returns an array containing the result.
template<class T> valarray<T> pow(const valarray<T> &*ob*, const T &*v*);	For all i, computes $ob1[i]^{v}$ and returns an array containing the result.
template<class T> valarray<T> sin(const valarray<T> &*ob*);	Obtains the sine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> sinh(const valarray<T> &*ob*);	Obtains the hyperbolic sine of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> sqrt(const valarray<T> &*ob*);	Obtains the square root of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> tan(const valarray<T> &*ob*);	Obtains the tangent of each element in *ob* and returns an array containing the result.
template<class T> valarray<T> tanh(const valarray<T> &*ob*);	Obtains the hyperbolic tangent of each element in *ob* and returns an array containing the result.

Table 37-4. *Transcendental Functions Defined for* ***valarray*** (continued)

The following program demonstrates a few of the many capabilities of **valarray**.

```
// Demonstrate valarray
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

int main()
{
  valarray<int> v(10);
  int i;

  for(i=0; i<10; i++) v[i] = i;

  cout << "Original contents: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  v = v.cshift(3);

  cout << "Shifted contents: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  valarray<bool> vb = v < 5;
  cout << "Those elements less than 5: ";
  for(i=0; i<10; i++)
    cout << vb[i] << " ";
  cout << endl << endl;

  valarray<double> fv(5);
  for(i=0; i<5; i++) fv[i] = (double) i;

  cout << "Original contents: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;
```

```
  fv = sqrt(fv);

  cout << "Square roots: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv + fv;
  cout << "Double the square roots: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv - 10.0;
  cout << "After subtracting 10 from each element:\n";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  return 0;
}
```

Its output is shown here:

```
Original contents: 0 1 2 3 4 5 6 7 8 9
Shifted contents: 3 4 5 6 7 8 9 0 1 2
Those elements less than 5: 1 1 0 0 0 0 0 1 1 1

Original contents: 0 1 2 3 4
Square roots: 0 1 1.41421 1.73205 2
Double the square roots: 0 2 2.82843 3.4641 4
After subtracting 10 from each element:
-10 -8 -7.17157 -6.5359 -6
```

The slice and gslice Classes

The **<valarray>** header defines two utility classes called **slice** and **gslice**. These classes encapsulate a slice (i.e., a portion) from an array. These classes are used with the subset forms of **valarray**'s operator[ ].

The **slice** class is shown here:

```
class slice {
public:
  slice();
  slice(size_t start, size_t len, size_t interval);
  size_t start() const;
  size_t size() const;
  size_t stride();
};
```

The first constructor creates an empty slice. The second constructor creates a slice that specifies the starting element, the number of elements, and the interval between elements (that is, the *stride*). The member functions return these values.

Here is a program that demonstrates **slice**.

```
// Demonstrate slice
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(10), result;
  unsigned int i;

  for(i=0; i<10; i++) v[i] = i;

  cout << "Contents of v: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[slice(0,5,2)];

  cout << "Contents of result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

The output from the program is shown here:

```
Contents of v: 0 1 2 3 4 5 6 7 8 9
Contents of result: 0 2 4 6 8
```

As you can see, the resulting array consists of 5 elements of **v**, beginning at 0, that are 2 apart.

The **gslice** class is shown here:

```
class gslice {
public:
  gslice();
  gslice()(size_t start, const valarray<size_t> &lens,
           const valarray<size_t> &intervals);
  size_t start() const;
  valarray<size_t> size() const;
  valarray<size_t> stride() const;
};
```

The first constructor creates an empty slice. The second constructor creates a slice that specifies the starting element, an array that specifies the number of elements, and an array that specifies the intervals between elements (that is, the *strides*). The number of lengths and intervals must be the same. The member functions return these parameters. This class is used to create multidimensional arrays from a **valarray** (which is always one-dimensional).

The following program demonstrates **gslice**.

```
// Demonstrate gslice()
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(12), result;
  valarray<size_t> len(2), interval(2);
  unsigned int i;

  for(i=0; i<12; i++) v[i] = i;

  len[0] = 3; len[1] = 3;
```

```
  interval[0] = 2; interval[1] = 3;

  cout << "Contents of v: ";
  for(i=0; i<12; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[gslice(0,len,interval)];

  cout << "Contents of result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

The output is shown here:

```
Contents of v: 0 1 2 3 4 5 6 7 8 9 10 11
Contents of result: 0 3 6 2 5 8 4 7 10
```

The Helper Classes

The numeric classes rely upon these "helper" classes, which your program will never instantiate directly: **slice_array**, **gslice_array**, **indirect_array**, and **mask_array**.

The Numeric Algorithms

The header **<numeric>** defines four numeric algorithms that can be used to process the contents of containers. Each is examined here.

accumulate

The **accumulate( )** algorithm computes a summation of all of the elements within a specified range and returns the result. Its prototypes are shown here:

```
template <class InIter, class T> T accumulate(InIter start, InIter end, T v);
template <class InIter, class T, class BinFunc>
        T accumulate(InIter start, InIter end, T v, BinFunc func);
```

Here, **T** is the type of values being operated upon. The first version computes the sum of all elements in the range *start* to *end*. The second version applies *func* to the running

total. (That is, *func* specifies how the summation will occur.) The value of *v* provides an initial value to which the running total is added.

Here is an example that demonstrates **accumulate( )**.

```
// Demonstrate accumulate()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5);
  int i, total;

  for(i=0; i<5; i++) v[i] = i;

  total = accumulate(v.begin(), v.end(), 0);

  cout << "Summation of v is: " << total;

  return 0;
}
```

The following output is produced:

```
Summation of v is: 10
```

adjacent_difference

The **adjacent_difference( )** algorithm produces a new sequence in which each element is the difference between adjacent elements in the original sequence. (The first element in the result is the same as the original first element.) The prototypes for **adjacent_difference( )** are shown here:

```
template <class InIter, class OutIter>
  OutIter adjacent_difference(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
  OutIter adjacent_difference(InIter start, InIter end, OutIter result,
                              BinFunc func);
```

Here, *start* and *end* are iterators to the beginning and ending of the original sequence. The resulting sequence is stored in the sequence pointed to by *result*. In the first form,

adjacent elements are subtracted, with the element at location *n* being subtracted from the element at location *n*+1. In the second, the binary function *func* is applied to adjacent elements. An iterator to the end of *result* is returned.

Here is an example that uses **adjacent_difference( )**.

```
// Demonstrate adjacent_difference()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(10), r(10);
  int i;

  for(i=0; i<10; i++) v[i] = i*2;
  cout << "Original sequence: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  adjacent_difference(v.begin(), v.end(), r.begin());

  cout << "Resulting sequence: ";
  for(i=0; i<10; i++)
    cout << r[i] << " ";

  return 0;
}
```

The output produced is shown here:

```
Original sequence: 0 2 4 6 8 10 12 14 16 18
Resulting sequence: 0 2 2 2 2 2 2 2 2 2
```

As you can see, the resulting sequence contains the difference between the value of adjacent elements.

inner_product

The **inner_product( )** algorithm produces a summation of the product of corresponding elements in two sequences and returns the result. It has these prototypes:

```
template <class InIter1, class InIter2, class T>
    T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v);
template <class InIter1, class InIter2, class T, class BinFunc1, class BinFunc2>
    T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v,
                    BinFunc1 func2, BinFunc2 func2);
```

Here, *start1* and *end1* are iterators to the beginning and end of the first sequence. The iterator *start2* is an iterator to the beginning of the second sequence. The value *v* provides an initial value to which the running total is added. In the second form, *func1* specifies a binary function that determines how the running total is computed, and *func2* specifies a binary function that determines how the two sequences are multiplied together.

Here is a program that demonstrates **inner_product( )**.

```
// Demonstrate inner_product()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v1(5), v2(5);
  int i, total;

  for(i=0; i<5; i++) v1[i] = i;
  for(i=0; i<5; i++) v2[i] = i+2;

  total = inner_product(v1.begin(), v1.end(),
                        v2.begin(), 0);

  cout << "Inner product is: " << total;

  return 0;
}
```

Here is the output:

```
Inner product is: 50
```

partial_sum

The **partial_sum( )** algorithm sums a sequence of values, putting the current total into each successive element of a new sequence as it goes. (That is, it creates a sequence that is a running total of the original sequence.) The first element in the result is the same as

the first element in the original sequence. The prototypes for **partial_sum( )** are shown here:

```
template <class InIter, class OutIter>
    OutIter partial_sum(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
    OutIter partial_sum(InIter start, InIter end, OutIter result,
                        BinFunc func);
```

Here, *start* and *end* are iterators to the beginning and end of the original sequence. The iterator *result* is an iterator to the beginning of the resulting sequence. In the second form, *func* specifies a binary function that determines how the running total is computed. An iterator to the end of *result* is returned.

Here is an example of **partial_sum( )**.

```
// Demonstrate partial_sum()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5), r(5);
  int i;

  for(i=0; i<5; i++) v[i] = i;
  cout << "Original sequence: ";
  for(i=0; i<5; i++)
    cout << v[i] << " ";
  cout << endl;

  partial_sum(v.begin(), v.end(), r.begin());

  cout << "Resulting sequence: ";
  for(i=0; i<5; i++)
    cout << r[i] << " ";

  return 0;
}
```

Here is its output:

```
Original sequence: 0 1 2 3 4
Resulting sequence: 0 1 3 6 10
```